

Sterling Hughes ([sterling@bumblebury.com](mailto:sterling@bumblebury.com)) – <http://www.edwardbear.org>

Harald Radi ([harald.radi@nme.at](mailto:harald.radi@nme.at)) – <http://www.nme.at>

# 1 ADT Abstract Data Types

*ADT, or the Abstract Data Types extension was released in its alpha form on Jan 21, 2003. This new extension, planned for inclusion in PHP5 will provide a base set of fully extensible data types, including Ordered Trees, Balanced Binary Trees, Graphs, Heaps, and Stacks. This paper covers ADT's current state of development, and planned future developments. It explains the usage of ADT, and the core algorithms and ideologies.*

## 1.1 Why Abstract Data Types

Object-oriented programming is often described as programming *abstract data types* and their relationships. Every written program is supposed to be a solution for a given problem, typically without a formal definition. Thus the first step is to create an abstract view or model of the problem.

This step, called *abstraction*, almost always leads to the very same subset of core problems, the only variations are their relations. This was the motivation that instead of reinventing the wheel each time, one could create a library that contains solutions for these very elemental problems in such a way, that they could not only used for very specific problem but also for a whole class of similar problems. Abstract data types were born.

For those of you with other backgrounds like C++, Java, C# etc. this concept is not new, the STL and various class libraries exist for a long time, but there was no comparable native support in PHP. Nevertheless PHP growing up and entering new application fields (php-cli, php-gtk, etc.) augmented the need for a set of flexible data types. PHP5 will finally contain most of the basic data types and their related operations known from other languages.

## 1.2 Implementation

As known from many other PHP extensions the ADT extension has a function API as well as an object-oriented API. There is no advantage on whether you are using one or the other, they are on a par.

It only depends on your affectation whether to use the function or the object-orientated API. You are even allowed, although not encouraged, to mix those paradigms.

To give you a better impression all examples are provided twice using the function and the object-oriented approach.

Some of the constructor functions take the function name of an element comparison function as an argument. These parameters are fully optional, if not provided PHP's default comparison functions (<, >, ==, !=) will be used.

As shown in the examples you can overload those to perform a meaningful comparison based on the type of the elements you store.

It is also important to mention, that all abstract data types are cloneable and that a clone is a deep copied replication of the original element.

### 1.2.1 Set

A set is an ordered container of elements that allows no duplicates. It provides set theoretical functions to intersect, unify, subtract and compare different sets. It also has functions to insert or remove elements to a set and to check the existence of an element in a given set.

Listing 1 and 2 show you how to perform basic operations with sets.

```
<?php
function comparator($a, $b) {
    if ($a < $b) return 1;
    if ($a > $b) return -1;
    return 0;
}

$set = adt_set_new("comparator");

adt_set_insert($set, "hello");
adt_set_insert($set, "hi");
adt_set_insert($set, "hallo");
adt_set_insert($set, "howdy");
adt_set_insert($set, "hey");

$set2 = adt_set_new("comparator");

adt_set_insert($set2, "grüßgott");
adt_set_insert($set2, "hoi");
adt_set_insert($set2, "hey");
adt_set_insert($set2, "hallo");
adt_set_insert($set2, "moin");

$clone = adt_set_clone($set2);

$intersect = adt_set_intersect($set, $set2);

if (adt_set_is_member($intersect, "hey")) {
    print "hurray\n";
}
```

```
if (adt_set_is_member($clone, "moin")) {  
    print "hurray\n";  
}  
?>
```

Listing 1: Set using function API

```
<?php  
function comparator($a, $b) {  
    if ($a < $b) return 1;  
    if ($a > $b) return -1;  
    return 0;  
}  
  
$set = new adt_set("comparator");  
  
$set->insert("hello");  
$set->insert("hi");  
$set->insert("hallo");  
$set->insert("howdy");  
$set->insert("hey");  
  
$set2 = new adt_set("comparator");  
  
$set2->insert("grüßgott");  
$set2->insert("hoi");  
$set2->insert("hey");  
$set2->insert("hallo");  
$set2->insert("moin");  
  
$clone = $set2->__clone();  
  
$intersect = $set->intersect($set2);  
  
if ($intersect->is_member("hey")) {  
    print "hurray";  
}  
  
if ($clone->is_member("moin")) {  
    print "hurray";  
}  
?>
```

Listing 2: Set using OO API

## 1.2.2 Binary Trees

For large amounts of input and more complex operations on that data a tree is often the chosen data structure for storing elements, because of its very cheap average element access cost.

The ADT extension provides two kinds of binary trees, a common binary tree and a binary search tree. The latter automatically sorts the element while inserting them and provides functions to access elements very quickly. The former is only a basic tree implementation and leaves the insertion strategy up to you.

```
<?php
function comparator($a, $b) {
    if ($a < $b) return 1;
    if ($a > $b) return -1;
    return 0;
}

$tree = adt_tree_new(ADT_TREE_TYPE_BINARY_SEARCH, "comparator");

adt_tree_insert($tree, 12);
adt_tree_insert($tree, 20);
adt_tree_insert($tree, 15);
adt_tree_insert($tree, 13);
adt_tree_insert($tree, 28);

var_dump(adt_tree_search($tree, 28, "comparator"));

adt_tree_traverse($tree, 0, "var_dump");
?>
```

Listing 3: Tree using function API

```
<?php
function comparator($a, $b) {
    if ($a < $b) return 1;
    if ($a > $b) return -1;
    return 0;
}

$t = new adt_tree(ADT_TREE_TYPE_BINARY_SEARCH, "comparator");

$t->insert(12);
$t->insert(20);
```

```
$t->insert(15);
$t->insert(13);
$t->insert(28);

var_dump($t->search(28, "comparator"));

$t->traverse(0, "var_dump");
?>
```

Listing 4: Tree using OO API

### 1.2.3 Heap

A heap is a *complete binary tree* meaning that there is no node without two sub nodes beside in the bottom level. Thus the subtree you would get after removing the bottom level is optimally balanced and therefore very shallow. Subsequent the costs for finding and accessing elements are very low.

Elements of a heap are topologically sorted, the root of the tree is always the smallest element with respect to the comparison function provided.

```
<?php
function comparator($a, $b) {
    if ($a < $b) return 1;
    if ($a > $b) return -1;
    return 0;
}

$heap = adt_heap_new(ADT_HEAP_TYPE_BINARY, "comparator");

adt_heap_insert($heap, "knorp");
adt_heap_insert($heap, "henk");
adt_heap_insert($heap, "zort");

$clone = adt_heap_clone($heap);

var_dump(adt_heap_extract($heap));
var_dump(adt_heap_extract($heap));

var_dump(adt_heap_extract($clone));
var_dump(adt_heap_extract($clone));
?>
```

Listing 5: Heap using function API

```
<?php
function comparator($a, $b) {
    if ($a < $b) return 1;
    if ($a > $b) return -1;
    return 0;
}

$heap = new adt_heap(ADT_HEAP_TYPE_BINARY, "comparator");

$heap->insert("knorp");
$heap->insert("henk");
$heap->insert("zort");

$clone = $heap->__clone();

var_dump($heap->extract());
var_dump($heap->extract());

var_dump($clone->extract());
var_dump($clone->extract());
?>
```

Listing 6: Heap using OO API

### 1.2.4 Stack

A stack is basically a single linked list which restricts element access to only the topmost element in that list. Another name for a stack is a LIFO (last in, first out) list.

Stacks are most useful when solving recursive problems in an iterative function, like parsing a XML file.

```
<?php
$stack = adt_stack_new();

adt_stack_push($stack, "hello");
adt_stack_push($stack, "world");

$clone = adt_stack_clone($stack);

var_dump(adt_stack_count($stack));
var_dump(adt_stack_count($clone));

var_dump(adt_stack_pop($stack));
```

```
var_dump(ad_t_stack_pop($stack));  
var_dump(ad_t_stack_pop($stack));  
?>
```

Listing 7: Stack using function API

```
<?php  
$stack = new ad_t_stack();  
  
$stack->push("hello");  
$stack->push("world");  
  
$clone = $stack->__clone();  
  
var_dump($stack->count());  
var_dump($clone->count());  
  
var_dump($stack->pop());  
var_dump($stack->pop());  
var_dump($stack->pop());  
?>
```

Listing 8: Stack using OO API

### 1.2.5 Queue

A queue, like a stack, is a single linked list. With a queue, however, elements are inserted at the top of the list but retrieved from its back. Therefore queues are also referenced as FIFO (first in, first out) lists.

Queues are useful to manage the communication in Producer/Consumer problems or wherever else elements are not processed the same time as they are produced, but the order of processing remains important.

```
<?php  
function comparator($a, $b) {  
    if ($a < $b) return 1;  
    if ($a > $b) return -1;  
    return 0;  
}  
  
$queue = ad_t_queue_new(ADT_QUEUE_TYPE_PRIORITY, "comparator");  
  
ad_t_queue_insert($queue, "zort");  
ad_t_queue_insert($queue, "narf");
```

## Implementation

---

```
$clone = adt_queue_clone($queue);  
  
var_dump(adt_queue_peek($queue));  
  
var_dump(adt_queue_extract($queue));  
  
var_dump(adt_queue_extract($clone));  
var_dump(adt_queue_extract($clone));  
?>
```

Listing 9: Queue using function API

```
<?php  
function comparator($a, $b) {  
    if ($a < $b) return 1;  
    if ($a > $b) return -1;  
    return 0;  
}  
  
$heap = new adt_heap(ADT_HEAP_TYPE_BINARY, "comparator");  
  
$heap->insert("knorp");  
$heap->insert("henk");  
$heap->insert("zort");  
  
$clone = $heap->__clone();  
  
var_dump($heap->extract());  
var_dump($heap->extract());  
  
var_dump($clone->extract());  
var_dump($clone->extract());  
?>
```

Listing 10: Queue using OO API

### 1.2.6 Graph

A graph by definition is a set of vertices that might be connected by a set of edges. A graph can contain either directed or undirected edges.

A direct edge means, that if it connects the vertex a with vertex b there is no connection back from b to a. An undirected edge means, that if a is connected with b, b is connected with a as well.

Graphs are useful to represent networks and perform operations of them like finding the shortest path from vertex a to b or to find the minimum spanning tree that connects all vertices by only using the existing edges.

```
<?php
function comparator($a, $b) {
    return ($a == $b);
}

$graph = adt_graph_new(ADT_GRAPH_TYPE_DIRECTED, "comparator");
adt_graph_insert_vertex($graph, "hello");
adt_graph_insert_vertex($graph, "howdy");

adt_graph_insert_edge($graph, "hello", "hi");
adt_graph_insert_edge($graph, "hello", "heya");
adt_graph_insert_edge($graph, "hello", "howdy");

adt_graph_insert_edge($graph, "howdy", "hello");
adt_graph_insert_edge($graph, "howdy", "servas");

$clone = adt_graph_clone($graph);
adt_graph_remove_edge($clone, "hello", "hi");

var_dump(adt_graph_adjacency_lists($graph));
var_dump(adt_graph_vertex_count($graph));
var_dump(adt_graph_edge_count($graph));

var_dump(adt_graph_adjacency_lists($clone));
var_dump(adt_graph_vertex_count($clone));
var_dump(adt_graph_edge_count($clone));
?>
```

Listing 11: Graph using function API

```
<?php
function comparator($a, $b) {
    return ($a == $b);
}

$graph = new adt_graph(ADT_GRAPH_TYPE_DIRECTED, "comparator");
$graph->insert_vertex("hello");
$graph->insert_vertex("howdy");

$graph->insert_edge("hello", "hi");
$graph->insert_edge("hello", "heya");
```

```
$graph->insert_edge("hello", "howdy");

$graph->insert_edge("howdy", "hello");
$graph->insert_edge("howdy", "servas");

$clone = $graph->__clone();
$clone->remove_edge("hello", "hi");

var_dump($graph->adjacency_lists());
var_dump($graph->vertex_count());
var_dump($graph->edge_count());

var_dump($clone->adjacency_lists());
var_dump($clone->vertex_count());
var_dump($clone->edge_count());
?>
```

Listing 12: Graph using OO API

### 1.3 Future Developments

Currently only the very basic data types are implemented by the extension. Limiting the user to only very few types is as amiss as giving him enough rope to shoot himself.

We yet have to determine the smallest possible set of types that provides enough flexibility to cover all practical issues.

After that we are heading against feature completeness, meaning that all types should support all the functions one would expect from them. This also means that we are working on a tighter integration into PHP. Data types will probably be enumerable with the foreach language construct and they should also be serializable sometime.

If we are still motivated enough after that we will go over all those algorithms again and try to replace the actually used straight forward ones by more optimized and improved versions.

And of course everyone is welcome to help.

### 1.4 References

- [WEI99] Data Structures & Algorithm Analysis in C++ Second Edition,  
Mark Allen Weiss, Adisson-Wesley
- [KNU97] The Art of Computer Programming,  
Donald E. Knuth, Adisson-Wesley